Capítulo DEZESSEIS

Operações de Stream em Coleções

Objetivos do Exame

- Desenvolver código que use métodos de dados de Stream e métodos de cálculo.
- Usar as interfaces java.util.Comparator e java.lang.Comparable.
- Ordenar uma coleção usando a API de Stream.

Comparator e Comparable

Para ordenar arrays ou coleções, o Java fornece duas interfaces muito semelhantes:

```java
java.util.Comparator
public interface Comparator<T> {
    int compare(T obj1, T obj2);
    // Outros métodos default e static ...
}

java.lang.Comparable
public interface Comparable<T> {
    int compareTo(T obj);
}
```

A diferença é que java.util.Comparator é implementada por uma classe que você usa para ordenar objetos de OUTRA classe, enquanto java.lang.Comparable é implementada pelo MESMO objeto que você deseja ordenar.

Com Comparator, você cria um objeto para comparar dois objetos de outro tipo a fim de ordená-los; é por isso que você recebe dois parâmetros, e o método se chama compare.

Com Comparable, você torna um objeto comparável a outro objeto do mesmo tipo para ordená-los; é por isso que você recebe apenas UM parâmetro e o método se chama compareTo. Como esta interface é mais fácil de compreender, vamos começar com ela.

---

java.lang.Comparable

O método a ser implementado é:

```java
int compareTo(T obj);
```

Como você pode ver, ele retorna um int. Aqui estão suas regras:

- Quando ZERO é retornado, significa que o objeto é IGUAL ao argumento.
- Quando um número MAIOR que zero é retornado, significa que o objeto é MAIOR que o argumento.
- Quando um número MENOR que zero é retornado, significa que o objeto é MENOR que o argumento.

Muitas classes do Java (como BigDecimal, BigInteger, wrappers como Integer, String, etc.) implementam essa interface com uma ordem natural (como 1, 2, 3, 4 ou A, B, C, a, b, c).

Como esse método pode ser usado para testar se um objeto é igual a outro, é recomendado que a implementação seja consistente com o método equals(Object) (se o método compareTo retorna 0, o método equals deve retornar true).

Uma vez que um objeto implementa essa interface, ele pode ser ordenado por Collections.sort() ou Arrays.sort(). Também pode ser usado como chave em um mapa ordenado (como TreeMap) ou em um conjunto ordenado (como TreeSet).

O seguinte é um exemplo de como um objeto pode implementar Comparable.

java Copiar Editar

```java
public class Computer implements Comparable<Computer> {
    private String brand;
    private int id;

    public Computer(String brand, int id) {
        this.brand = brand;
        this.id = id;
    }

    // Vamos comparar primeiro pela marca e depois pelo id
    public int compareTo(Computer other) {
        // Reutilizando a implementação de String
        int result = this.brand.compareTo(other.brand);

        // Se os objetos forem iguais, comparar pelo id
        if(result == 0) {
            // Vamos fazer a comparação "manualmente"
            if(this.id > other.id) result = 1;
            else if(this.id < other.id) result = -1;
            // else result = 0;
        }
        return result;
    }

    // equals e compareTo devem ser consistentes
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Computer)) return false;
        return this.brand.equals(other)
            && this.id == ((Computer)other).id;
    }

    public static void main(String[] args) {
        Computer c1 = new Computer("Lenovo", 1);
        Computer c2 = new Computer("Apple", 2);
        Computer c3 = new Computer("Dell", 3);
        Computer c4 = new Computer("Lenovo", 2);

        // Algumas comparações
        System.out.println(c1.compareTo(c1)); // c1 == c1
        System.out.println(c1.compareTo(c2)); // c1 > c2
        System.out.println(c2.compareTo(c1)); // c2 < c1
        System.out.println(c1.compareTo(c4)); // c1 < c2
        System.out.println(c1.equals(c4));    // c1 != c2

        // Criando uma lista e ordenando
        List<Computer> list = Arrays.asList(c1, c2, c3, c4);
        Collections.sort(list);
        list.forEach(
           c -> System.out.format("%s-%d\n", c.brand, c.id)
        );
    }
}
```

Ao executar este programa, esta é a saída:

```diff
0
11
-11
-1
false
Apple-2
Dell-3
Lenovo-1
Lenovo-2
```

---

java.util.Comparator

O método a ser implementado é:

```java
int compare(T o1, T o2);
```

As regras do valor retornado são semelhantes às de Comparable:

- Quando ZERO é retornado, significa que o PRIMEIRO argumento é IGUAL ao SEGUNDO argumento.
- Quando um número MAIOR que zero é retornado, significa que o PRIMEIRO argumento é MAIOR que o SEGUNDO argumento.
- Quando um número MENOR que zero é retornado, significa que o PRIMEIRO argumento é MENOR que o SEGUNDO argumento.

Uma vantagem de usar um Comparator em vez de Comparable é que você pode ter muitos Comparators para ordenar o mesmo objeto de diferentes formas.

Por exemplo, podemos pegar a classe Computer do exemplo anterior para criar um Comparator que ordena primeiro pelo id e depois pela brand, e como as regras do valor retornado são praticamente as mesmas de Comparable, podemos usar o método compareTo:

```java
Comparator<Computer> sortById =
    new Comparator<Computer>() {
        public int compare(Computer c1, Computer c2) {
            int result = Integer.compare(c1.id, c2.id);
            return result == 0
                ? c1.brand.compareTo(c2.brand) : result;
        }
    };
```

Além disso, Integer.compare(x, y) é equivalente a:

```java
Integer.valueOf(x).compareTo(Integer.valueOf(y));
```

Felizmente, Comparator é uma interface funcional, então podemos usar uma expressão lambda em vez de uma classe interna:

```java
Comparator<Computer> sortById = (c1, c2) -> {
    int result = Integer.compare(c1.id, c2.id);
    return result == 0
        ? c1.brand.compareTo(c2.brand) : result;
};
```

Então, quando a usamos na lista do exemplo anterior:

```java
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list, sortById);
list.forEach(
    c -> System.out.format("%d-%s\n", c.id, c.brand)
);
```

A saída é:

```
1-Lenovo
2-Apple
2-Lenovo
3-Dell
```

Caso esteja se perguntando, Comparable também é considerado uma interface funcional, mas como espera-se que Comparable seja implementado pelo objeto sendo comparado, você quase nunca a usará como uma expressão lambda.

Métodos Default e Static em Comparator

No Java 8, com a introdução de métodos default e static em interfaces, temos alguns métodos úteis em Comparator para simplificar nosso código como:

```java
Comparator<T> Comparator.comparing(Function<? super T, ? extends U>)
Comparator<T> Comparator.comparingInt(ToIntFunction<? super T>)
Comparator<T> Comparator.comparingLong(ToLongFunction<? super T>)
Comparator<T> Comparator.comparingDouble(ToDoubleFunction<? super T>)
```

Esses métodos recebem uma Function (uma expressão lambda) que retorna o valor de uma propriedade do objeto que será usada para criar um Comparator usando o valor retornado por compareTo (observe também as versões para tipos primitivos).

Por exemplo:

```java
Comparator<Computer> sortById =
    Comparator.comparing(c -> c.id);
```

Ou:

```java
Comparator<Computer> sortById =
    Comparator.comparingInt(c -> c.id);
```

Eles são equivalentes a:

```java
Comparator<Computer> sortById = new Comparator<Computer>() {
    public int compare(Computer c1, Computer c2) {
        return Integer.valueOf(c1.id)
                      .compareTo(Integer.valueOf(c2.id));
    }
};
```

---

thenComparing

Outro método útil é thenComparing, que encadeia dois Comparators (observe que este **não é um método static**):

```java
Comparator<T> thenComparing(Comparator<? super T>)
Comparator<T> thenComparing(Function<? super T, ? extends U>)
Comparator<T> thenComparingInt(ToIntFunction<? super T>)
Comparator<T> thenComparingLong(ToLongFunction<? super T>)
Comparator<T> thenComparingDouble(ToDoubleFunction<? super T>)
```

Dessa forma, podemos simplificar o código para criar um Comparator que ordena por id e depois por brand usando:

```java
Comparator<Computer> sortByIdThenByBrand =
    Comparator.comparing((Computer c) -> c.id)
              .thenComparing(c -> c.brand);
```

Por fim, o método default reversed() criará um Comparator que inverte a ordem do Comparator original:

```java
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list,
    Comparator.comparing((Computer c) -> c.id).reversed());
list.forEach(
    c -> System.out.format("%d-%s\n", c.id, c.brand));
```

A saída:

```
3-Dell
2-Apple
2-Lenovo
1-Lenovo
```

Ordenando um Stream

Ordenar um *stream* é simples. O método:

```java
Stream<T> sorted()
```

Retorna um *stream* com os elementos ordenados de acordo com sua **ordem natural**. Por exemplo:

```java
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
list.stream()
    .sorted()
    .forEach(System.out::println);
```

Irá imprimir:

```
2
37
38
54
57
```

O único requisito é que os elementos do *stream* implementem java.lang.Comparable (dessa forma, eles são ordenados em ordem natural). Caso contrário, uma ClassCastException poderá ser lançada.

Se quisermos ordenar usando uma ordem diferente, há uma versão deste método que recebe um java.util.Comparator (essa versão **não está disponível** para *streams* primitivos como IntStream):

```java
Stream<T> sorted(Comparator<? super T> comparator)
```

Por exemplo:

```java
List<String> strings =
    Arrays.asList("Stream", "Operations", "on", "Collections");
strings.stream()
    .sorted((s1, s2) -> s2.length() - s1.length())
    .forEach(System.out::println);
```

Ou:

```java
List<String> strings =
    Arrays.asList("Stream", "Operations", "on", "Collections");
strings.stream()
    .sorted(Comparator.comparing(
              (String s) -> s.length()).reversed())
    .forEach(System.out::println);
```

Ambos irão imprimir:

```
Collections
Operations
Stream
on
```

O primeiro trecho de código retornará um valor positivo se o comprimento da primeira String for menor que o da segunda, e um valor negativo caso contrário, para ordenar as Strings em **ordem decrescente**.

O segundo trecho de código criará um Comparator com o comprimento da String em ordem natural (**ordem crescente**) e depois inverterá essa ordem.

Métodos de Dados e de Cálculo

A interface Stream fornece os seguintes métodos de dados e cálculo:

```java
long count()
Optional<T> max(Comparator<? super T> comparator)
Optional<T> min(Comparator<? super T> comparator)
```

E, no caso das versões primitivas da interface Stream, temos os seguintes métodos:

IntStream

```java
OptionalDouble average()
long count()
OptionalInt max()
OptionalInt min()
int sum()
```

LongStream

```java
OptionalDouble average()
long count()
OptionalLong max()
OptionalLong min()
long sum()
```

DoubleStream

```java
OptionalDouble average()
long count()
OptionalDouble max()
OptionalDouble min()
double sum()
```

- count() retorna o número de elementos no *stream* ou zero se o *stream* estiver vazio:

```java
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
System.out.println(list.stream().count()); // 5
```

- min() retorna o valor mínimo no *stream* envolto em um Optional ou um Optional vazio se o *stream* estiver vazio.

- max() retorna o valor máximo no *stream* envolto em um Optional ou um Optional vazio se o *stream* estiver vazio.

Quando falamos sobre primitivos, é fácil saber qual é o valor mínimo ou máximo. Mas quando estamos falando sobre objetos (de qualquer tipo), o Java precisa saber **como compará-los** para saber qual é o máximo e o mínimo. É por isso que a interface Stream precisa de um Comparator para max() e min():

```java
List<String> strings =
    Arrays.asList("Stream", "Operations", "on", "Collections");
strings.stream()
    .min(Comparator.comparing(
            (String s) -> s.length())
    ).ifPresent(System.out::println); // on
```

sum() retorna a soma dos elementos no *stream* ou zero se o *stream* estiver vazio:

```java
System.out.println(
    IntStream.of(28, 4, 91, 30).sum()
); // 153
```

average() retorna a média dos elementos no *stream* envolta em um OptionalDouble ou um OptionalDouble vazio se o *stream* estiver vazio:

```java
System.out.println(
    IntStream.of(28, 4, 91, 30).average()
); // 38.25
```

Pontos-chave

- java.util.Comparator é implementada por uma classe que você usa para ordenar objetos de OUTRA classe.
- java.lang.Comparable é implementada pelo MESMO objeto que você deseja ordenar.

Os principais métodos de ambas as interfaces retornam um int. Suas regras são muito semelhantes:

- Quando ZERO é retornado, significa que o objeto (ou primeiro argumento) é IGUAL ao (segundo) argumento.
- Quando um número MAIOR que zero é retornado, significa que o objeto (ou primeiro argumento) é MAIOR que o (segundo) argumento.
- Quando um número MENOR que zero é retornado, significa que o objeto (ou primeiro argumento) é MENOR que o (segundo) argumento.

Os métodos comparing(), thenComparing() e reversed() são métodos auxiliares da interface Comparator adicionados no Java 8.

O método sorted() da interface Stream retorna um *stream* com os elementos ordenados de acordo com sua ordem natural. Você também pode passar um Comparator como argumento.

- count() retorna o número de elementos no *stream* ou zero se o *stream* estiver vazio.
- min() retorna o menor valor no *stream* envolto em um Optional ou um vazio se o *stream* estiver vazio.
- max() retorna o maior valor no *stream* envolto em um Optional ou um vazio se o *stream* estiver vazio.

- sum() retorna a soma dos elementos no *stream* ou zero se o *stream* estiver vazio.
- average() retorna a média dos elementos no *stream* envolta em um OptionalDouble ou vazio se o *stream* estiver vazio.

Autoavaliação

1. Dado:

```java
public class Question_16_1 {
    public static void main(String[] args) {
        List<String> strings =
            Arrays.asList("Stream", "Operations", "on", "Collections");
        Collections.sort(strings, String::compareTo);
        System.out.println(strings.get(0));
    }
}
```

Qual é o resultado?

A. Collections
B. on
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

2. Qual das seguintes instruções retorna um Comparator válido?

A. (String s) -> s.length();
B. Comparator.reversed();
C. Comparator.thenComparing((String s) -> s.length());
D. Comparator.comparing((String s) -> s.length() * -1);

---

3. Dado:

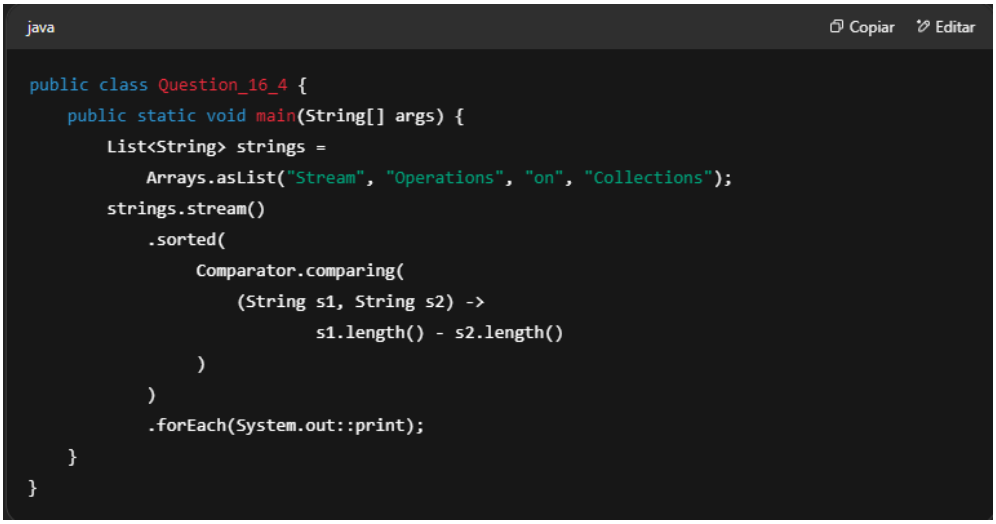
```java
public class Question_16_3 {
    public static void main(String[] args) {
         List<Integer> list = Arrays.asList(30, 5, 8);
         list.stream().max().get();
    }
}
```

Qual é o resultado?

A. 5
B. 30
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

4. Dado:



Qual é o resultado?

A. CollectionsOperationsStreamOn
B. onStreamOperationsCollections
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução